# 1. Gestionnaire de fichier en FLASH

## 1.1 Utilisation de la mémoire FLASH

### 1.1.1 Principe de fonctionnement de la mémoire FLASH

La mémoire FLASH est une zone de mémoire capable de conserver des données en l'absence d'alimentation. Elle sert à mémoriser de manière durable des informations importantes du système et des applications tournant dans le décodeur.

Si la mémoire FLASH est accessible librement en lecture, l'écriture, plus complexe, et soumise à certaines contraintes :

- Il n'est pas possible de faire passer un bit de la valeur zéro à la valeur un sans effacer (mettre à un) une page entière. La taille d'une page varie selon les mémoires FLASH. Elles sont généralement de 16 ou 64 Kilo-octets.
- La mise à zéro d'une page prend un temps non négligeable de l'ordre de la seconde.
- Même une écriture ne faisant passer que des bits de la valeur un à zéro ne peut se faire par un accès direct à la mémoire FLASH. Il faut passer par les *Devices* qui positionnent la mémoire en mode d'écriture avant de lui envoyer (de manière séquentielle ?) les nouvelles valeurs des octets.

Cette mémoire servant à mémoriser des informations parfois indispensables au bon fonctionnement de l'IRD, il est nécessaire d'assurer sa cohérence. En particulier il faut s'assurer que l'on peut toujours revenir à un état stable à la mise sous tension même en cas de coupure brutale et impromptue de l'alimentation.

### 1.1.2 Utilisation de la mémoire FLASH dans le RUN

Le RUN écrit toujours dans la mémoire FLASH page par page en utilisant une page tampon. A chaque fois que l'on veut écrire un fichier dans la mémoire FLASH on cherche une page avec assez de place libre pour mémoriser le contenu du fichier. Cette page peut être celle contenant la valeur précédente du fichier qui sera remplacée par la nouvelle valeur ou une nouvelle page si le fichier est nouveau. La page tampon est effacée. Les données valides de la page choisie sont recopiées dans la page tampon. La page choisie devient la nouvelle page tampon.

Les informations concernant la cohérence des données (numéro de la page tampon actuelle, CRC des pages) sont mémorisées dans une EEPROM.

Cette utilisation comporte plusieurs inconvénients :

- L'effacement de la page tampon se fait juste avant l'écriture (à vérifier) d'où perte de temps. Il serait préférable d'effectuer l'effacement de la nouvelle page tampon juste après l'écriture dans l'ancienne.
- Il est nécessaire d'avoir en mémoire RAM un buffer de même taille que la page à écrire.
- Nécessité d'écrire toute la page d'un coup.
- La présence d'une EEPROM est indispensable.

### 1.1.3 Utilisation de la mémoire FLASH dans la MVM

On utilise la possibilité, offerte dans la nouvelle API, d'écrire un certain nombre d'octets à un endroit encore vierge d'une page de mémoire FLASH. Tant qu'il reste possible d'écrire dans une page qui n'est pas pleine on rajoute des blocs à la fin cette page. Les blocs précédents sont invalidés en positionnant à zéro certain bits de l'en-tête des blocs.

La MVM garde une page tampon qui est utilisée, lorsque toutes les autres pages sont saturées, pour recopier une page contenant beaucoup de données invalides en supprimant ces données invalides. La nouvelle page tampon est toujours effacée immédiatement après une recopie.

Les informations de cohérence sont mémorisées dans les pages elles-mêmes.

Les avantages de cette utilisation sont :

- La plupart du temps il est juste nécessaire d'écrire le bloc que l'on veut modifier et non toute la page.
- La plupart du temps il n'est pas nécessaire d'attendre l'effacement la page tampon avant d'écrire (sauf si l'on doit écrire plusieurs pages successivement).
- Même pendant la recopie d'une page vers la page tampon, il est possible de copier les données bloc par bloc. Il est par conséquent uniquement nécessaire de réserver en mémoire RAM la taille du plus grand bloc.

# 1.2 Le format des fichiers dans la mémoire FLASH

Chaque page de la mémoire FLASH se compose d'un en-tête et de plusieurs blocs consécutifs de données. Un fichier sera composé de un ou plusieurs blocs répartis dans une ou plusieurs pages. Un répertoire se compose d'un seul bloc. Chaque fichier ou répertoire possède un identificateur unique. Chaque fichier ou répertoire fait référence, par son identificateur, au répertoire auquel il appartient.

## 1.2.1 Format et cohérence des pages dans la mémoire FLASH

Les informations concernant la cohérence des données en mémoire FLASH seront mémorisées dans la mémoire FLASH elle-même. Pour les pages de la mémoire FLASH l'en-tête décrit ses états possibles au cours de sa « vie » :

- Page vierge (*Empty*).
- Page en cours de création (*Write*) et numéro de la page qui est en cours de recopie ici.
- Recopie terminée (*New*) et numéro de la page qui vient d'être recopiée ici.
- Page valide (*Valid*)
- Page invalide (*Invalid*)

Ces états sont codés par des valeurs numériques qui permettent toujours de passer d'un état à un état ultérieur. Le premier état (*Empty*) est obtenu en effaçant la page. L'en-tête de la page contient aussi un « *magic number* » comme vérification supplémentaire de la validité de la page. Des octets inutilisés sont insérés pour que chaque champ commence à une adresse multiple de sa taille et que chaque bloc commence sur un multiple de la plus grande taille de champ possible : celle du mot long (8 octets).

Les états *Write*, *New* et *Invalid* sont des états temporaires utilisés pendant la recopie d'une page vers la page tampon qui est la phase la plus critique. Ces états permettent de restaurer la cohérence à la mise sous tension même en cas d'arrêt pendant la copie.

Un état stable des pages en mémoire est caractérisé par la présence d'au moins une page dans l'état *Empty* et d'autres pages à l'état *Valid*.

Une recopie d'une page vers la page tampon à partir d'un état stable passe par les étapes suivantes :

1. Passage de la page tampon de l'état *Empty* à l'état *Write*. Le numéro de la page source est mémorisé dans l'en-tête de la page tampon.
2. Recopie des données utiles de la page source vers la page tampon. Cette recopie peut se faire bloc par bloc. Cependant le passage à l'état *New* ne se fera que lorsque tous les blocs valides de la page source auront été copiés ou remplacés par de nouvelles versions.
3. Passage de la page tampon de l'état *Write* à l'état *New*.
4. Passage de la page source de l'état *Valid* à l'état *Invalid*.
5. Passage de la page tampon de l'état *New* à l'état *Valid*. Il ne s'agit plus d'une page tampon.
6. Effacement de la page source. Passage à l'état *Empty*.

S'il arrive à la mise sous tension qu'une page et une seule ne soit pas dans l'état *Empty* ou *Valid*, il y a eu arrêt pendant la copie. On peut restaurer un état stable en fonction de l'état de cette page :

- La page est dans l'état *Write* : on l'efface. Elle passe à l'état *Empty*. Les données en cours de copie sont perdues. Les données précédentes sont toujours présentes et valides.
- La page est dans l'état *New* : on invalide la page qui vient d'être copiée (état *Valid* vers état *Invalid*), puis on valide la page courante (état *New* vers état *Valid*), enfin on efface la page qui vient d'être copiée (état *Invalid* vers état *Empty*).
- La page est dans l'état *Invalid* : on l'efface. Elle passe à l'état *Empty*.

Il ne peut pas arriver que plus d'une page soit dans un état différent de l'état *Empty* ou *Valid*. Si cela se produisait malgré tout, il faudrait conclure à une corruption de la mémoire FLASH et prendre des mesures radicales (effacement de toute la mémoire FLASH ?).

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

La Figure 1 montre le format général d'une page de la mémoire FLASH et donne les valeurs numériques correspondant aux différents états de la page.

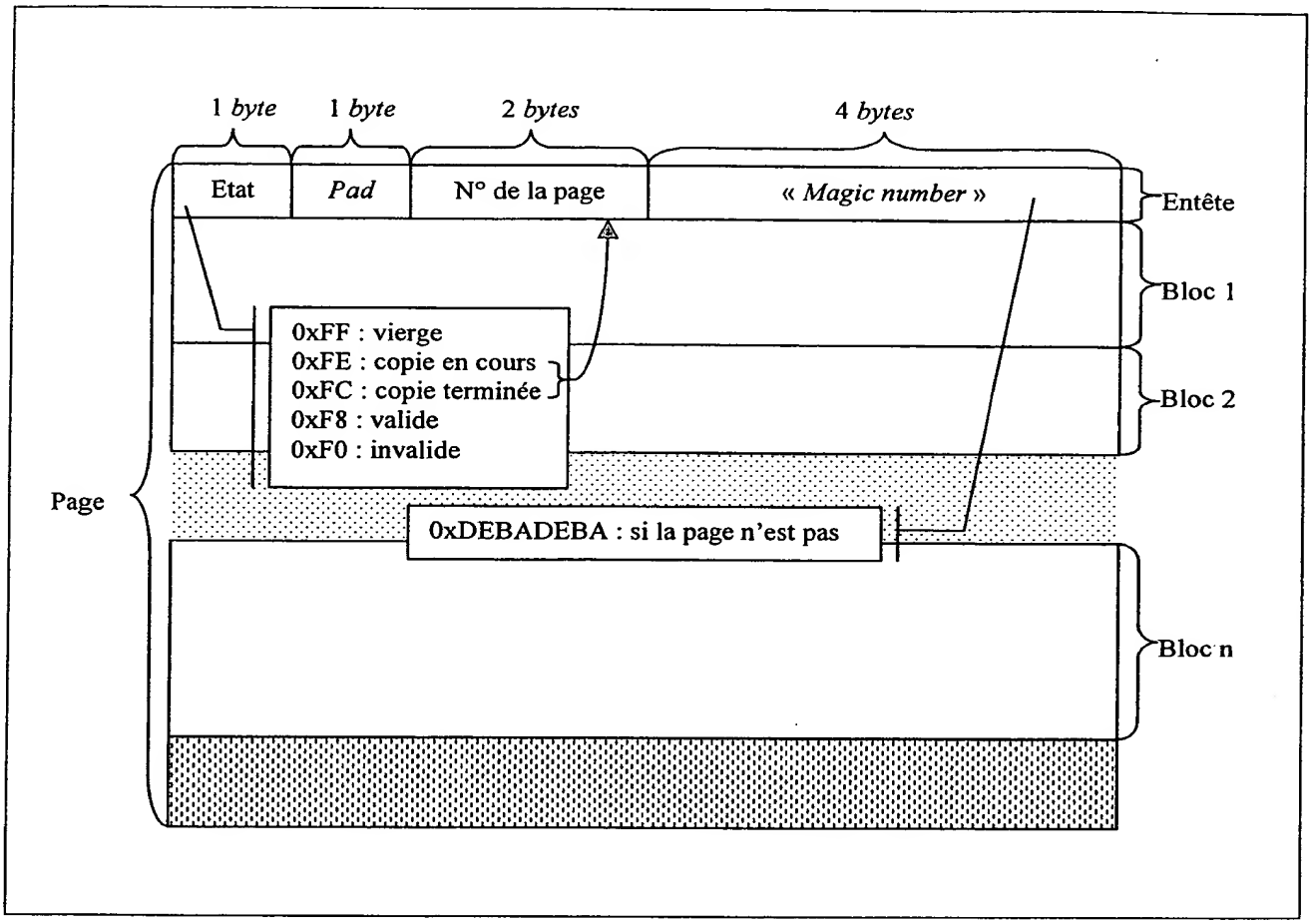


**Figure 1 : Format des pages de la mémoire FLASH**

## 1.2.2 Format et cohérence des fichiers dans la mémoire FLASH

Chaque fichier est composé de un ou plusieurs blocs. Un bloc est une zone de la mémoire FLASH contenue dans une page. Chaque bloc comprend un en-tête permettant de gérer sa cohérence et de recomposer le fichier auquel il appartient.

Comme une page un bloc peut avoir plusieurs états au cours de sa « vie » :

- Bloc libre (*Free*). Seul le dernier bloc d'une page peut être dans cet état. Il représente toute la partie de la page encore utilisable pour des écritures de nouveaux blocs.
- Bloc en cours de création (*Create*). L'en-tête du bloc (y compris sa taille) est en cours d'écriture. La taille du bloc contenue dans l'en-tête peut ne pas être encore valide.
- Bloc en cours d'écriture (*Write*). L'en-tête du bloc donc sa taille est valide. Les données peuvent ne pas être encore valides.
- Bloc nouvellement validé (*New*). Le bloc est prêt à être utilisé si tous les autres blocs à modifier pour ce même fichier sont aussi nouvellement validé.
- Bloc valide (*Valid*). L'ancienne version du bloc, si elle existait, a été invalidée. Le bloc sera utilisé à la prochaine utilisation du fichier.
- Bloc invalide (*Invalid*). Une nouvelle version de ce bloc a été créée ou le fichier à été effacé.

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

L'en-tête de chaque bloc contient un octet pour mémoriser l'état du bloc. Cet octet sert aussi à mémoriser les informations suivantes qui concernent plutôt l'état du fichier auquel appartient le bloc :

- Ce bloc est le bloc d'indice le plus élevé écrit pour ce fichier (bit *Ending*).
- Ce bloc appartient à un fichier en cours d'écriture (bit *Open*). Seul un bloc *Ending* peut aussi être *Open*.
- Ce bloc appartient à un répertoire (bit *Directory*).

L'en-tête d'un bloc contient aussi l'identificateur du fichier auquel il appartient et l'index du bloc parmi les blocs composant le fichier. Combinés avec l'état de chaque bloc, ces informations permettent de restaurer la cohérence de chaque fichier même en cas d'arrêt durant la copie.

Enfin l'en-tête contient la taille des données du bloc ce qui permet de trouver le bloc suivant.

Chaque entrée de l'arbre est composée d'au moins un bloc. Le premier bloc ne contient pas les données de l'entrée mais des informations sur l'entrée. Ces informations sont :

- L'identificateur du répertoire auquel appartient cette entrée. Cette information permet de recréer l'arborescence des fichiers.
- Un octet décrivant les attributs de l'entrée (fichier ou répertoire, ...).
- Un octet décrivant les modes d'accès autorisés sur ce fichier pour le propriétaire, pour le groupe et pour les autres.
- Deux octets identifiant le propriétaire du fichier
- Deux octets identifiant le groupe du fichier
- Faut-il mémoriser la date de création et la date de modification du fichier ?
- Le nom du fichier terminé par un code ASCII de zéro.

Un répertoire peut être vu comme un fichier ne contenant que ce premier bloc.

La Figure 2 ci-dessous le contenu du premier bloc d'un fichier ou d'un répertoire avec la description de l'en-tête.

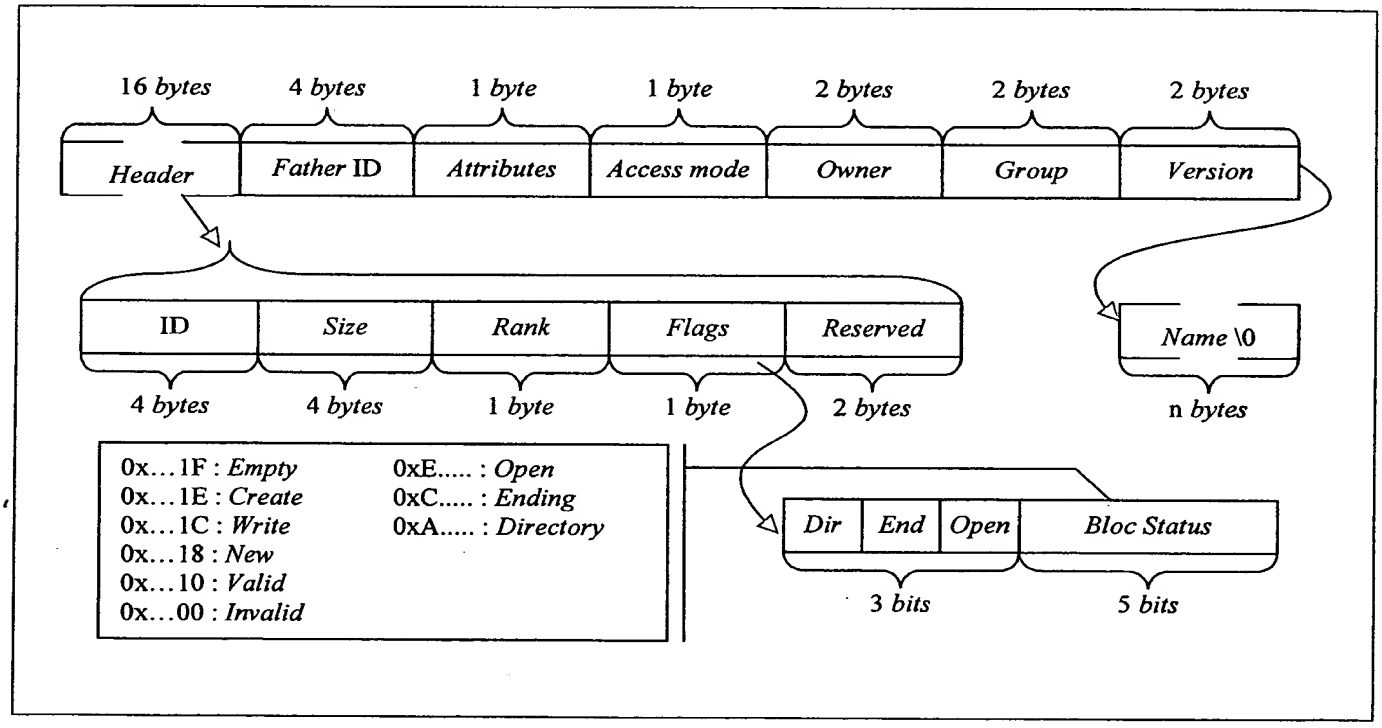


**Figure 2 : En-tête et données du premier bloc d'un fichier ou d'un répertoire**

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

#### *1.2.2.1 Ecriture d'un répertoire ou d'un fichier composé d'un seul bloc*

C'est le cas le plus facile. On enchaîne les étapes suivantes :

1. Trouver, à la fin d'une page, un bloc libre capable de contenir le bloc à écrire. Il est dans l'état *Free*.
2. Création de l'en-tête du nouveau bloc. Passage du bloc a l'état *Create*, les bits *Ending* et *Open* restent à 1, puis écriture de l'en-tête : identificateur du fichier, indice du bloc dans le fichier (bloc numéro 1), taille du bloc.
3. Passer le bloc à l'état *Write*. Ecriture des données du bloc.
4. Passer le bloc à l'état *New*. Cet état permet d'attendre que tous les autres blocs du fichier soit écrits et les anciennes versions des blocs invalidés. Dans le cas d'un fichier monobloc c'est un état très temporaire.
5. Passer le bit *Open* à zéro. Cette étape correspond à la fermeture du fichier. A partir de maintenant les nouvelles données seront validées même en cas d'arrêt et de redémarrage du système.
6. Passer l'ancienne version du bloc, si elle existe, à l'état *Invalid*.
7. Passer la nouvelle version du bloc à l'état *Valid*.

#### *1.2.2.2 Ecriture d'un fichier composé de plusieurs blocs*

On exécute plusieurs fois une partie des étapes utilisées pour un fichier contenant un seul bloc. La méthode est la suivante :

- Les étapes 1 à 4 sont effectuées par indice croissant des blocs modifiés du fichier. Si un bloc est identique à la version existante il n'est pas nécessaire de créer une nouvelle version.
  A chaque nouveau bloc, passer les bits *Ending* et *Open* du bloc *New* précédemment terminal à zéro si le nouveau bloc à un indice supérieur.
- Si la fermeture du fichier demande la troncature, le bit *Open* du bloc *New* et *Ending* est passé à zéro, validant tous les nouveaux blocs.
- Si la fermeture du fichier ne demande pas la troncature, le bit *Ending* du bloc *New* d'indice le plus élevé est passé à zéro.
- Le bit *Open* du bloc *New* d'indice le plus élevé est passé à zéro, validant tous les nouveaux blocs.
- L'étape 6 est effectuée par indice croissant sur les anciennes versions de tous les blocs qui ont été modifiés.
- L'étape 7 est effectuée par indice croissant sur les nouvelles versions de tous les blocs qui ont été modifiés.

#### *1.2.2.3 Restauration de l'intégrité de la mémoire FLASH*

Examinons comment, en utilisant les étapes d'écriture décrites ci-dessus, il est possible de restaurer l'intégrité de la mémoire FLASH même en cas d'arrêt durant n'importe laquelle de ces étapes.

Au démarrage de l'IRD, la mémoire FLASH est parcourue page par page. Suivant l'état de chaque bloc on effectue l'une des opérations suivantes :

- Le bloc est dans l'état *Free*. C'est très bien, on le laisse tranquille et on peut passer à la page suivante.
- Le bloc est dans l'état *Create*. L'en-tête du bloc n'est pas dans un état cohérent. En particulier la taille des données contenue dans l'en-tête peut être invalide. On laisse le bloc dans cet état. Le bloc suivant commence juste derrière l'en-tête de celui-ci.
- Le bloc est dans l'état *Write*. La taille du bloc contenue dans l'en-tête est valide. Les données sont en cours d'écriture donc partiellement invalides. Le bloc passe à l'état *Invalid*. Le bloc suivant commence après les données de celui-ci : il faut avancer de la taille de l'en-tête plus la taille des données.
- Le bloc est dans l'état *New*. Le bloc est mémorisé dans la liste des blocs du fichier dont l'identificateur est celui contenu dans l'en-tête. Ce bloc sera validé ou non lors d'une deuxième passe de vérification qui sera effectuée sur chaque fichier. Le bloc suivant commence après les données de celui-ci.
- Le bloc est dans l'état *Valid*. Même traitement que pour un bloc dans l'état *New*.
- Le bloc est dans l'état *Invalid*. On le laisse tranquille. Le bloc suivant commence après les données de celui-ci.

Cette première passe à permit d'invalider les blocs à l'état *Write* et surtout de mémoriser tous les blocs *Valid* ou *New* de chaque fichier. Il s'agit maintenant de décider quels blocs utiliser pour chaque fichier. On se trouve confronté à plusieurs configurations différentes que l'on va vérifier l'une après l'autre :

- Le fichier contient un bloc à l'état *New* et *Open*. Une création de fichier était en cours mais elle n'est pas terminée. Il faut faire passer tous les blocs *New* du fichier à l'état *Invalid*.
- Le fichier n'a pas de bloc *Ending* et contient au moins un bloc *Valid*. Théoriquement impossible : la mémoire FLASH est corrompue. Il faut faire passer tous les blocs du fichier à l'état *Invalid* et détruire le fichier.

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

- Il existe un bloc *Ending Valid* et au moins un bloc *Valid* avec un indice supérieur ou il existe plus d'un bloc *Valid* de même indice. Théoriquement impossible : la mémoire FLASH est corrompue. Il faut faire passer tous les blocs du fichier à l'état *Invalid* et détruire le fichier.
- Il existe un bloc *Ending New* et au moins un bloc *New* avec un indice supérieur ou il existe plus d'un bloc *New* de même indice. Théoriquement impossible : la mémoire FLASH est corrompue. Il faut faire passer tous les blocs à l'état *New* du fichier à l'état *Invalid* puis continuer les vérifications suivantes.
- Les indices de blocs du fichier ne sont pas contigus : ils ne commencent pas à zéro ou il existe des indices intermédiaires manquant. Il faut faire passer tous les blocs *New* du fichier à l'état *Invalid*. Si les blocs *Valid* restant ne sont pas non plus contigus, la mémoire FLASH est corrompue : il faut faire passer tous les blocs du fichier à l'état *Invalid* (donc détruire le fichier).
- Il existe des blocs *New* mais pas de bloc *Open*. Une écriture vient de se terminer. Une nouvelle version de certains blocs du fichier a été écrite avec succès. Les anciennes versions des blocs sont en cours d'invalidation. S'il existe un bloc *New Ending* et un bloc *Valid Ending*, la fermeture avait demandé la troncature : le bit *Ending* du bloc *Valid* doit être passé à zéro. Ensuite il faut poursuivre l'invalidation par indice croissant des blocs *Valid* ayant un bloc *New* de même indice. Enfin, il faut valider par indice croissant tous les blocs *New*.
- Tous les blocs du fichier sont *Valid*. Un seul bloc terminal existe. Son indice est n. Aucun bloc n'a un indice supérieur à n. Tous les indices entre 1 et n sont présent une et une seule fois. C'est le cas idéal dans lequel tous les fichiers encore existant doivent se trouver après la phase de vérification en cours.

### 1.2.3 Cohérence de l'arbre des fichiers dans la mémoire FLASH

Il reste à vérifier que tous les fichiers et répertoires dans la mémoire FLASH sont bien reliés à la racine. Pour cela il faut vérifier que chaque répertoire est relié à la racine (identificateur zéro) en remontant l'arbre de manière récursive. Si à un moment de la remontée le père d'un répertoire est introuvable ou n'est pas un répertoire (ce dernier cas est théoriquement impossible), le répertoire doit être détruit (son bloc invalidé).

Après le parcours sur tous les répertoires, il faut vérifier que chaque fichier a bien pour père un répertoire existant. Si ce n'est pas le cas le fichier doit être détruit (tous ses blocs invalidés).

## 1.3 Fonctionnalités du gestionnaire de mémoire FLASH

Le gestionnaire de mémoire FLASH est appelé par le gestionnaire de fichier. Les fonctionnalités de base qu'il doit fournir sont :

- La création au démarrage de l'IRD d'un arbre contenant les répertoires et les fichiers présents dans la mémoire FLASH. Cette création se fait en appelant la fonction **FSCreateEntry** du gestionnaire de fichier.
- La création ou l'ouverture en écriture d'un fichier de la mémoire FLASH, l'écriture dans ce fichier et sa fermeture. Ces dernières fonctionnalités peuvent modifier la position de certains fichiers ou répertoires dans la mémoire FLASH. Elles doivent donc modifier les informations du gestionnaire de fichier avec la fonction **FSChangeEntry**.

## 1.4 Interface du gestionnaire de mémoire FLASH

Le gestionnaire de mémoire FLASH doit travailler en collaboration avec le gestionnaire de fichiers. En particulier Il doit être capable de générer un arbre représentant les fichiers de la mémoire FLASH utilisable par le gestionnaire de fichiers. Une solution est de fournir au gestionnaire de mémoire FLASH les structures des nœuds et des branches de l'arbre pour qu'il puisse le créer. Une autre solution est que le gestionnaire de fichier fournisse des fonctions pour ajouter un répertoire ou un fichier dans l'arbre. Cette deuxième solution semble préférable car elle limite l'interdépendance entre les deux gestionnaires et évite la duplication de code.

### 1.4.1 Les types de données

#### *1.4.1.1 FlashOpenedFile*

**NOM**

**FlashOpenedFile** – Structure opaque décrivant un fichier de la mémoire FLASH en cours d'écriture

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

**SYNOPSYS**

```
typedef      struct _FlashOpenedFile FlashOpenedFile;
```

**DESCRIPTION**

Pour pouvoir créer ou écrire une entrée (fichier ou répertoire) dans la mémoire FLASH, le gestionnaire de fichier doit appeler l'une des fonctions **FlashCreateEntry** ou **FlashOpenEntry**. Ces deux fonctions retournent un pointeur vers une structure opaque utilisée par le gestionnaire de mémoire FLASH lors des écritures qui suivront.

## 1.4.2 Les fonctions

Les fonctions décrites ci-dessous sont susceptibles de déplacer des fichiers dans la mémoire FLASH. Elles peuvent donc appeler à tout moment la fonction du gestionnaire de fichier permettant de mettre à jour les informations sur les fichiers en mémoire FLASH : **FSChangeEntry**.

### *1.4.2.1 FlashCreateTree*

**NOM**

**FlashCreateTree** – Crée l'arbre de fichiers de la mémoire FLASH

**SYNOPSIS**

```
Int     FlashCreateTree(FSDirEntry *theRootEntry);
```

**DESCRIPTION**

Le paramètre est un pointeur vers le répertoire racine de la mémoire FLASH vue par le gestionnaire de fichier.

Parcourt la mémoire FLASH et crée un arbre contenant l'arborescence des fichiers présents dans la mémoire FLASH. L'arbre est créé en appelant la fonction **FSAddEntry** du gestionnaire de fichiers en lui passant le pointeur vers un répertoire déjà créé par le gestionnaire de fichier et les données de la nouvelle entrée.

**CODE DE RETOUR**

Le code de retour est **ERR_NONE** si tout s'est bien passé.

### *1.4.2.2 FlashCreateEntry*

**NOM**

**FlashCreateEntry** – Crée une entrée (fichier ou répertoire) dans la mémoire FLASH

**SYNOPSIS**

```
FlashOpenedFile *
        FlashCreateEntry( char        *theFatherName,
                          char        *theEntryName,
                          UInt16      theOwner,
                          Uint16      theGroup,
                          Uint8       theAccessMode,
                          Uint8       theAttributes);
```

**DESCRIPTION**

Crée une entrée dans la mémoire FLASH. L'entrée doit ne pas exister (aucun test n'est effectué).

Le nom du répertoire père doit être un pointeur vers le champ du premier bloc d'un répertoire contenant le nom du répertoire. Partant de ce pointeur, il est possible de retrouver l'identificateur du père qui se trouve un certain nombre d'octets avant.

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

**CODE DE RETOUR**

Le code de retour est **NULL** en cas d'erreur ou si l'entrée est un répertoire.

### *1.4.2.3 FlashChangeEntry*

**NOM**

**FlashChangeEntry** – Modifie une entrée (fichier ou répertoire) dans la mémoire FLASH

**SYNOPSIS**

```
Int     FlashChangeEntry( char        *theOldFatherName,
                          char        *theOldName,
                          char        *theNewFatherName,
                          char        *theNewName,
                          UInt16      theOwner,
                          Uint16      theGroup,
                          Uint8       theAccessMode,
                          Uint8       theAttributes);
```

**DESCRIPTION**

Modifie les informations sur une entrée de la mémoire FLASH. L'entrée doit déjà exister. Elle ne doit pas être en cours d'écriture.

**theOldFatherName, theOldName** et **theNewFatherName** sont des pointeurs vers des noms en mémoire FLASH contenus dans le premier bloc des entrées correspondantes. Les identificateurs de ces entrées se trouvent quelques octets avant.

**theNewName** est un pointeur vers le nouveau nom. Ce n'est pas forcément un pointeur en mémoire FLASH. Il sera recopié en mémoire FLASH dans le nouveau premier bloc qui va être créé.

**CODE DE RETOUR**

Le code de retour est **ERR_NONE** si tout s'est bien passé.

### *1.4.2.4 FlashDeleteEntry*

**NOM**

**FlashDeleteEntry** – Détruit une entrée dans la mémoire FLASH

**SYNOPSIS**

```
Int FlashDeleteEntry(char *theEntryName);
```

**DESCRIPTION**

Le nom sert à retrouver l'identificateur (quelques octets avant le nom).

**CODE DE RETOUR**

Le code de retour est **ERR_NONE** si tout s'est bien passé.

### *1.4.2.5 FlashOpenFile*

**NOM**

**FlashOpenFile** – Ouvre un fichier en écriture dans la mémoire FLASH

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

**SYNOPSIS**

```
FlashOpenedFile *FlashOpenFile(char *theFileName);
```

**DESCRIPTION**

Prépare une structure interne qui sera utilisée pour les écritures dans ce fichier jusqu'à la fermeture.

Le nom du fichier est un pointeur vers le champ contenant le nom dans le premier bloc du fichier. L'identificateur du fichier se trouve quelques octets avant.

**CODE DE RETOUR**

Le code de retour est NULL en cas de problème.

### *1.4.2.6 FlashWriteFile*

**NOM**

**FlashWriteFile** – Ecrit dans un fichier ouvert en mémoire FLASH

**SYNOPSIS**

```
Int FlashWriteFile(     FlashOpenedFile   *theOpenedFile,
                        UInt32            theIndex,
                        char              *theBuffer,
                        UInt32            theLength);
```

**DESCRIPTION**

Ecrit dans un fichier ouvert dans la mémoire FLASH dans le bloc dont l'indice est donné en paramètre, les données du buffer dont la taille est donnée en paramètre. Si aucun bloc d'indice supérieur n'a été écrit depuis que le fichier à été ouvert :

1) le précédent bloc *New* terminal, s'il existe, est marqué comme non terminal,
2) le bloc courant reste marqué comme terminal.

**CODE DE RETOUR**

**ERR_NONE** si tout se passe bien.

### *1.4.2.7 FlashCloseFile*

**NOM**

**FlashCloseFile** – Ferme un fichier en écriture dans la mémoire FLASH

**SYNOPSIS**

```
Int FlashCloseFile(FlashOpenedFile *theOpenedFile, Bool truncate);
```

**DESCRIPTION**

Ferme un fichier ouvert en écriture. Si le paramètre **truncate** est *true*, le bloc de plus grand index parmi ceux qui viennent d'être écrit reste le bloc terminal. Si ce paramètre est *false*, l'ancien bloc terminal reste terminal.

**CODE DE RETOUR**

**ERR_NONE** si tout se passe bien.

**ERR_NO_TERMINAL** si **truncate** est *true* et aucun bloc n'a été écrit ou si **truncate** est *false* et le fichier est nouveau.

© Copyright CANAL + 1996 - 1997

Ce document ne peut être communiqué à des tiers sans autorisation écrite préalable des ayants droits.

THIS PAGE BLANK (USPTO)